Shortest Path For Google Foobar (Prepare The Bunnies Escape)

I have been working on Google Foobar since a few days ago. I am currently in the third level but is stuck in the second challenge of "Prepare the Bunnies' Escape."

I have checked this post but it did not fully resolve my problem in Google Foobar as it is still giving me `Time limit exceeded` errors.

For reference, here is the challenge.

> ## Prepare the Bunnies' Escape
>
> You have maps of parts of the space station, each starting at a prison exit and ending at the door to an escape pod. The map is represented as a matrix of 0s and 1s, where 0s are passable space and 1s are impassable walls. The door out of the prison is at the top left $(0, 0)$ and the door into an escape pod is at the bottom right $(w-1, h-1)$.
>
> Write a function `answer(map)` that generates the length of the shortest path from the prison door to the escape pod, where you are allowed to remove one wall as part of your remodeling plans. The path length is the total number of nodes you pass through, counting both the entrance and exit nodes. The starting and ending positions are always passable (0). The map will always be solvable, though you may or may not need to remove a wall. The height and width of the map can be from 2 to 20. Moves can only be made in cardinal directions; no diagonal moves are allowed.
>
> ## Test cases
>
> Input:
>
> ```
> maze = [[0, 1, 1, 0], [0, 0, 0, 1], [1, 1, 0, 0], [1, 1, 1, 0]]
> ```
>
> Output:
>
> ```
> 7
> ```
>
> Input:
>
> ```
> maze = [[0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1,
> 1], [0, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0]]
> ```
>
> Output:
>
> ```
> 11
> ```

What I have done is create a graph based on the matrix and apply the shortest path algorithm (I am not sure what the algorithm is exactly called).

The code I have below runs and gives me the proper length of the shortest path.

```
class Node(object):
    def __init__(self, identity, x, y):
        self.neighbours = []
        self.identity = identity
        self.coordinates = (x, y)

    def addNeighbour(self, node_key):
        self.neighbours.append(node_key)

class Graph(object):
    def __init__(self, matrix):
        self.matrix = matrix
        self.passable_walls = set()
```

```
        self.nodes = {}

    def create(self):
        length = len(self.matrix)
        for row in range(length):
            for col in range(length):
                identity = 0
                if self.matrix[row][col] == 1:
                    identity = 1

                node = Node(identity, row, col)

                # Get the neighbours
                if self._hasNeighbour(row - 1, col): # Is the top cell there?
                    node.addNeighbour((row - 1, col))

                if self._hasNeighbour(row + 1, col): # Is the bottom cell there?
                    node.addNeighbour((row + 1, col))

                if self._hasNeighbour(row, col - 1): # Is the left cell there?
                    node.addNeighbour((row, col - 1))

                if self._hasNeighbour(row, col + 1): # Is the right cell there?
                    node.addNeighbour((row, col + 1))

                self.nodes[(row, col)] = node

    def getWalls(self):
        length = len(self.matrix)
        traversed_walls = 0
        for row in range(length):
            for col in range(length):
                if self.matrix[row][col] == 1:
                    self.passable_walls.add((row, col))

    def findShortestPath(self, start, end, passable_wall, path=set()):
        path.add(start)
        if start == end:
            return path

        if start not in self.nodes:
            return None

        shortest = None
        for node in self.nodes[start].neighbours:
            if node not in path and (self.nodes[node].identity == 0 or
self.nodes[node].coordinates == passable_wall):
                curr_path = set(path)
                new_path = self.findShortestPath(self.nodes[node].coordinates, end,
passable_wall, curr_path)
                if new_path:
                    if not shortest or len(new_path) < len(shortest):
                        shortest = new_path

        return shortest

    def _hasNeighbour(self, row, col):
        length = len(self.matrix)
        if row < 0 or col < 0 or row >= length or col >= length:
            return False

        return True

def answer(matrix):
    graph = Graph(matrix)

    shortest = None
    matrix_length = len(matrix)
    start_node = (0, 0)
    exit_node = (matrix_length - 1, matrix_length - 1)
    graph.create()
    graph.getWalls()
    for wall in graph.passable_walls:
        curr_path = graph.findShortestPath(start_node, exit_node, wall, path=set())
        if curr_path is not None:
            if shortest is None:
                shortest = len(curr_path)
            else:
                if len(curr_path) < shortest:
                    shortest = len(curr_path)

    return shortest
```

When I run it using this line

```
print(answer([[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
              [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
              [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
              [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
              [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]))
```

It takes `0.992` seconds to get the shortest path length according to cProfile. Using cProfile reveals that the bottleneck is the shortest path algorithm (`def findShortestPath(...)`).

Is there a problem with my algorithm? How could I improve it? What would be a better approach?

python programming-challenge time-limit-exceeded pathfinding

edited Apr 13 at 12:40
Community♦
1

asked Jan 21 at 17:05
Sean Francis N. Bal
189 6

2 Your algorithm is inefficient. You are essentially doing backtracking, but this has exponential complexity and you need to use some quicker method like BFS to solve it efficiently. – Raziman T V Jan 21 at 18:39

1 Answer

Basically you need a breadth-first search with a minor tweak:

1. each node represents a cell in the grid (x- and y-coordinates),
2. each node knows its "saldo" (how many walls it may penetrate).

What comes to that *saldo*, if a node has zero saldo, it may not generate those its neighbors that are occupied by wall. If saldo is $s > 0$, and the node has a wall neighbor node u, u is generated with saldo $s - 1$.

The rest is breadth-first search: as soon you remove the exit node from the queue, just print its distance from the starting node:

```
import time
from collections import deque


class Node:

    def __init__(self, x, y, saldo, grid):
        self.x = x
        self.y = y;
        self.saldo = saldo
        self.grid = grid

    def __hash__(self):
        return self.x ^ self.y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def get_neighbors(self):
        neighbors = []
        x = self.x
        y = self.y
        saldo = self.saldo
        grid = self.grid
        rows = len(grid)
        columns = len(grid[0])

        if x > 0:
            wall = grid[y][x - 1] == 1
            if wall:
                if saldo > 0:
                    neighbors.append(Node(x - 1, y, saldo - 1, grid))
            else:
                neighbors.append(Node(x - 1, y, saldo, grid))

        if x < columns - 1:
            wall = grid[y][x + 1] == 1
            if wall:
                if saldo > 0:
                    neighbors.append(Node(x + 1, y, saldo - 1, grid))
            else:
                neighbors.append(Node(x + 1, y, saldo, grid))

        if y > 0:
            wall = grid[y - 1][x] == 1
            if wall:
                if saldo > 0:
                    neighbors.append(Node(x, y - 1, saldo - 1, grid))
            else:
```

```
                neighbors.append(Node(x, y - 1, saldo, grid))

        if y < rows - 1:
            wall = grid[y + 1][x]
            if wall:
                if saldo > 0:
                    neighbors.append(Node(x, y + 1, saldo - 1, grid))
            else:
                neighbors.append(Node(x, y + 1, saldo, grid))

        return neighbors


class GridEscapeRouter:

    def __init__(self, grid, saldo):
        self.grid = grid
        self.rows = len(grid)
        self.columns = len(grid[0])
        self.saldo = saldo

    def get_escape_route_length(self):
        source = Node(0, 0, self.saldo, self.grid)
        queue = deque([source])
        distance_map = {source: 1}

        while queue:
            current_node = queue.popleft()

            if current_node.x == self.columns - 1 and\
                current_node.y == self.rows - 1:
                return distance_map[current_node]

            for child_node in current_node.get_neighbors():
                if child_node not in distance_map.keys():
                    distance_map[child_node] = distance_map[current_node] + 1
                    queue.append(child_node)

        return 1000 * 1000 * 1000 # Cannot escape

def milliseconds():
    return int(round(time.time() * 1000))


start_time = milliseconds()
router = GridEscapeRouter(
    [[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
    1)

route_length = router.get_escape_route_length();
end_time = milliseconds()

print "Route length", route_length, "in", end_time - start_time, "milliseconds."

router = GridEscapeRouter([[0, 1, 1, 0],
                           [0, 0, 0, 1],
                           [1, 1, 0, 0],
                           [1, 1, 1, 0]],
                          1)
print router.get_escape_route_length()

router = GridEscapeRouter([[0, 0, 0, 0, 0, 0],
                           [1, 1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 0, 0],
                           [0, 1, 1, 1, 1, 1],
                           [0, 1, 1, 1, 1, 1],
                           [0, 0, 0, 0, 0, 0]],
                          1)

print router.get_escape_route_length()
```

Prints:

```
Route length 39 in 15 milliseconds.
7
11
```

answered Jan 21 at 19:29

coderodde
12.1k 4 14 68

1 I feel so dumb for not figuring out the second minor tweak you mentioned. Thanks for this! You need more upvotes for this answer. – Sean Francis N. Ballais Jan 22 at 2:57

This algorithm tries to find the shortest path allowing for 1 wall from the top left corner to the bottom right corner. So it should return the same result if you try to find the shortest path from the bottom right to the top left. But if you make "source" in get_escape_route_length be Node(self.columns-1, self.rows-1, self.saldo, self.grid) and make the if condition "if current_node.x == 0 and current_node.y ==0:" instead, the code returns nothing... (continued below) – Katie Dobbs Mar 25 at 9:40

When I trace the last node it ends at, it's at x=3, y=0. It starts from(19,19), goes left until (13,19), uses the wall up at (13,18), eventually gets to (3,0) and then it's trapped. It can't go left or down because of walls, can't go back right (that's where it came from) and can't go up (it's on the top row). When I try to verify my solution that used this algorithm, it said 2 of the 5 test cases failed and I suspect it's related to this observation. – Katie Dobbs Mar 25 at 9:42

Here's the path it take's (in reverse order). (3, 0) (4, 0) (5, 0) (6, 0) (7, 0) (8, 0) (9, 0) (10, 0) (11, 0) (12, 0) (13, 0) (14, 0) (15, 0) (16, 0) (17, 0) (18, 0) (19, 0) (19, 1) (19, 2) (18, 2) (17, 2) (16, 2) (15, 2) (14, 2) (13, 2) (12, 2) (11, 2) (10, 2) (9, 2) (8, 2) (7, 2) (6, 2) (5, 2) (4, 2) (4, 3)(4, 4) (4, 5) (5, 5) (6, 5) (6, 6) (6, 7) (7, 7) (7, 8) (8, 8) (8, 9) (8, 10)(9, 10) (10, 10) (10, 11) (10, 12) (10, 13) (11, 13) (11, 14) (11, 15) (12, 15) (13, 15) (13, 16) (13, 17) (13, 18) [WALL] (13, 19) (14, 19) (15, 19) (16, 19) (17, 19) (18, 19) (19,19) – Katie Dobbs Mar 25 at 9:49

I just found this other question on this site that used your code but found a test case where it fails. (To be clear, I'm not trying to pile anything up on you, I'm just providing you with some data in case it helps you see the bug). Thanks. – Katie Dobbs Mar 25 at 9:53

@Katie Dobbs Thanks, I will take a look! – coderodde Mar 25 at 12:10

Some observations: It seems the algorithm works fine when all the moves on the shortest path are down or right, in which case the answer is 2*len(maze) - 1. For the 20x20 example in your code, it's 39. For the 4x4 it's 7 and for the 6x6 it's 11. But when you need do something else like turn back up, the algorithm doesn't return anything. E.g. for this, where the answer should be 10: [[0, 1, 1, 0, 1, 0, 0], [0, 0, 0, 0, 1, 1, 0]] – Katie Dobbs Mar 25 at 14:04

Thanks for @coderodde's answer. It brings a good idea on how to solve this problem. But there is a small bug in the function: def __eq__(self, other) the return value should also consider the 'saldo' value of each node. So the return value should be like this: self.x == other.x and self.y == other.y and self.saldo == other.saldo Thanks again. This answer helped me on passing all 5 tests. – RLi Apr 10 at 12:37